LISP Interpreter Report

Casey Haynes
casey.haynes@student.nmt.edu
New Mexico Institute of Mining and Technology
Department of Computer Science
Socorro, New Mexico, 87801

ABSTRACT

The LISP interpreter project is a great way to learn about recursion and other language design facets covered in CSE 324. I wrote my LISP interpreter in Java, since I have not yet worked with C++ enough to write an interpreter with it.

KEYWORDS

LISP Interpreter, Java, Recursion

1 IMPLEMENTATION LEVEL

Status Reporting Table 1

LISP Construct	Status
Variable Reference	Partially Done
Constant literal	Done
Quotation	Done
Conditional	Not Done
Variable Definition	Partially Done
Function Call	Not Done
Assignment	Not Done
Function Definition	Not Done
Arithmetic operators on integer type	Done
"car" and "cdr"	Done
The built-in function: "cons"	Partially Done
Sqrt, exp	Partially Done
>, <, ==, <=, >=, !=	Not Done

Status Reporting Table 2

Extra Credit LISP Construct	Status
BIG-Integer on +, -, *	Not Implemented
Compressed car, cdr	Not Implemented
"mapcar" built-in function	Not Implemented
"Lambda" built-in function	Not Implemented

2 IMPLEMENTATION DETAILS BEHIND EACH EXPRESSION

- Variable Reference:
  Variables are not fully implemented, however, I have an abstract class called "expression" which contains attributes "value" and "type" which allows for assignment to the data "type", meaning that if a variable class were to be created, variables could be included in the interpreter with a data structure, whether that's an ArrayList or something more efficient.
- Constant Literal:
  The LISP interpreter runs "Interpreter.java" as it's main driver class, where an input string is taken in. Once data is input, it is augmented with spaces, and is split on spaces and put into an array of strings. For constant literal, if the first character in the input excluding opening parentheses is not a defined quote or operator atom, the input's value is printed to the output file. It's the default case in the main switch statement of the class "Tokenizer."
- Quotation:
  The LISP interpreter runs "Interpreter.java" as it's main driver class, where an input string is taken in. Once data is input, it is augmented with spaces, and is split on spaces and put into an array of strings. Quotation is implemented by reading in the input data, and directing the control flow to a quote case of the main switch statement in the class "Tokenizer." For Quotation, if a quote is detected, the text following it is treated as data.
- Conditional:
  Not Implemented
- Variable Definition:
  Not Implemented
- Function Call:
  Not Implemented
- Assignment:
  Not Implemented
- Function Definition:
  Not Implemented
- Arithmetic Ops on int:
  The LISP interpreter runs "Interpreter.java" as it's main driver class, where an input string is taken in. Once data is input, it is augmented with spaces, and is split on spaces and put into an array of strings. The class "ArithmeticOp" handles the control flow for Arithmetic Operations. Depending on the symbol detected, control is sent to a different case in the switch statement. Essentially, if a symbol is detected, a parse tree is populated, with the operator being the root node of the tree, and the numbers given the children. The tree is then walked though, and the expression is evaluated depending on the operator (Each op has different behavior as defined in a switch statement in ArithmeticOp class)
- car and cdr
  The LISP interpreter runs "Interpreter.java" as it's main driver class, where an input string is taken in. Once data is input, it is augmented with spaces, and is split on spaces and put into an array of strings. If the expression "car" or "cdr" is detected, the control flow is sent to cases in a switch statement within the "Tokenizer" method in the driver class. It parses for quotes or parentheses and mimics car or cdr

accordingly. Note, recursive implementations do not work due to a fatal flaw somewhere in my parse tree apparatus.
- cons
  The LISP interpreter runs "Interpreter.java" as it's main driver class, where an input string is taken in. Once data is input, it is augmented with spaces, and is split on spaces and put into an array of strings. Cons operates largely in the same capacity as car and cdr. It is not yet working, due to a NullPointerException being thrown somewhere in the program execution.
- Sqrt, exp
  The LISP interpreter runs "Interpreter.java" as it's main driver class, where an input string is taken in. Once data is input, it is augmented with spaces, and is split on spaces and put into an array of strings. Works similarly to the other mathematic operators
- >, <, ==, <=, >=, !=
  Not Implemented

3 CLASS TOPICS RELEVANT TO THE IMPLEMENTATION

By far, the most important class topic used for implementing the LISP interpreter was the idea of recursion. LISP is essentially recursive by nature (nested parentheses). Second was the concept of a syntax/parse tree. Recursion and the tree data structure go hand in hand regarding the LISP implementation, since the tree is populated with expressions and atoms, and the deeper the level of recursion, the deeper the level of the resultant subtree. Furthermore, the discussion of syntax v. semantics was something I had never learned in any previous computer science course, which was incredibly important for completing the interpreter, as one must sift through different sorts of data in a singular string based on the context.

- Variable Reference:
  Syntax v. semantics.
- Constant Literal:
  Data Structuring / Activation Records
- Quotation:
  Syntax v. Semantics
- Conditional:
  Syntax v. Semantics
- Variable Definition:
  Data Structuring / Parsing Information
- Function Call:
  Data Structuring / Parsing Information
- Assignment:
  Data Structuring / Parsing Information
- Function Definition:
  Data Structuring / Parsing Information
- Arithmetic Ops on int:
  Parse Tree / Recursion
- car and cdr
  Syntax v. Semantics / Recursion
- cons
  Syntax v. Semantics / Recursion
- Sqrt, exp
  Data Structuring / Parsing Information
- >, <, ==, <=, >=, !=
  Recursion / Data Structuring / Parsing Information

4 IMPLEMENTION DETAILS NOT IN THE PREVIOUS CATEGORIES

Because I wrote the interpreter in Java, the concepts of Object Orientation and Encapsulation were especially important. In order for the parse tree to work with multiple data types, the tree had to be generic, and the data types themselves were containerized multiple times. I'm a novice OOP programmer, so encapsulation ultimately was my downfall. I had the tree built and populating correctly, but I could not get data back out it with an abstracted evaluation class. As such, the tree took most of my time and took away time to complete other LISP related items.

5 OTHER MENTIONS

The data structure which this project was built around was the syntax / parse tree. Ultimately, I could not get a final implementation to work with recursion, even though that's the entire reason I chose to work with a tree to begin with. Essentially, If I could fix my tree and fully comprehend the level of encapsulation I employed, I could fix the Achilles' Heel of this project. As such, more practice with trees and recursion is needed for me to write a LISP interpreter that fully works.

6 FINAL THOUGHTS

My main problem with the project was a lack of time to complete it. I am currently taking Systems Programming, Intro to Java Programming, Basic Concepts of Mathematics, and Principles of Programming Languages simultaneouly, which severely limited the number of hours I was able to work on the project. Also, I chose to use a tree structure in order to better facilitate recursion, as we learned in PPL, but my encapsulation scheme went awry with the tree, so no recursion ended up functioning. We had enough time to work on this project, but if I had more hours in each day I could have fixed these issues.

Also worth noting, I implemented my interpreter in Java because I didn't feel I had enough experience in C++ to implement a full interpreter. I learned a lot about the maintainability of Java and good Software Engineering practices during this assignment, because I nearly lost control of my code at multiple points becuase of how Byzantine it got. Another important point, my implementation relies heavily on the "Expression" abstract class and the "Tokenizer" method in the Interpreter class. See code comments for more information. [1][2][3]

REFERENCES

[1] Shadron Gudmunson. [n. d.]. Consulation on the Project.
[2] jbytecode. 2015. Lets implement a simple Lisp Interpreter in Java. Retrieved May 1, 2019 from http://stdioe.blogspot.com/2012/01/lets-implement-simple-lisp-interpreter.html
[3] jtanderson. 2012. LispInterpreter. Retrieved May 1, 2019 from https://github.com/jtanderson/LispInterpreter